# Functions, procedures and methods

The *"Don't Repeat Yourself"* principle requires that a code using two or more times the same construct should be abstracted into a common name.

With constants this was seen before, one can simply give a name to the constant value and reuse this name instead of the value throughout the rest of the code.

The question becomes tricky when it is not a constant but a full computation that should be abstracted, for example:

```
double p1 = 320;
System.out.println(p1 * VAT / 100);

double p2 = 45;
System.out.println(p2 * VAT / 100);

double p3 = 65.30;
System.out.println(p3 * VAT / 100);
```

Here the entire pattern formed by "`System.out.println(X * VAT / 100)`" should be abstracted and it is not a simple constant. This is where it becomes time to define a *function* instead.

## Functions: the mathematical view

In math we *define* a function for example like this:

$$f(x) = x + 1, \quad \forall x \in \mathbb{R}$$

or:

$$f : \forall x \in \mathbb{R}, f(x) = (x + 1) \in \mathbb{R}$$

or:

$$\begin{array}{rcl} f : \mathbb{R} & \mapsto & \mathbb{R} \\ f : x & \mapsto & x + 1 \end{array}$$

When we do this we say "*f* is a function from $\mathbb{R}$ to $\mathbb{R}$, which for every *x* equals *x+1*". And when we say this, we specify 5 things:

- the *name* of the function: *f*
- the *parameter* of the function: *x*

- the *domain* of the function (set of the parameter): $\mathbb{R}$
- the *expression* of the function: *x+1*
- the *image* or *range* of the function (set of the result): $\mathbb{R}$

The previous example has the same set as domain and image, but this is not always so. For example:

$$\begin{array}{lcl} g : \mathbb{N} \times \mathbb{N}^* & \mapsto & \mathbb{Q}^+ \\ g : n, m & \mapsto & \frac{n}{m} \end{array}$$

This function g has two parameters that are natural numbers, the second parameter must be non-zero, and associates every pair of parameters to the positive fractional number resulting from the division of one by the other.

Once functions are defined in math, it is possible to *use* them via the notation of "application", for example "f(3)" or "g(1,2)" denote respectively the application of $f$ on the value 3, and the application of $g$ on the pair (1, 2).

## Simple functions: math functions in code

Each programming language has its own way to define and use functions in the mathematical sense. In Java, we use the following general construct:

Syntax:

> <type> <name> `(` [ <type> <name> [ `,` <type> <name> ]* ] `)` `{` <statements...> `}`
>
> (a type followed by a name followed by an opening parenthesis, followed by zero or more occurrences of a type followed by a name, separated by commas, followed by a closing parenthesis)

Semantics:

> Defines a function with the name defined on the left before the opening parenthesis, with the parameters defined within the parentheses, with the various domains and image set describes by data types. The image is given by the type on the far left, whereas the domain set(s) are given by the type(s) between parentheses. Every time the function is applied (see below), the statements given on the right will be executed again.
>
> Furthermore, within the "<statements...>" part on the right, the special construct is recognized:
>
> Syntax:
>
> > `return` <value> `;`
> >
> > (the keyword "`return`" followed by a value followed by a semicolon)
>
> Semantics:

When this point is reached during execution, the value on the right becomes the value for the entire function and the execution of the function stops.

To use this, the following general method applies:

1. ensure you have a good understanding of which mathematical function you want to encode.
2. select a name for the function. Write this, followed by an opening and closing parenthesis.
3. between the parenthesis, write the name of the parameters.
4. in front of every name, select a type that describes the domain of the parameter (see below).
5. in front of the name of the function, select a type that describes the image of the function.
6. encode the expression of the function between the braces `{` and `}`.

For example, the functions $f$ and $g$ presented above can be encoded as follows:

```
int f(int x)
{
   return x + 1;
}

double g(int n, int m)
{
   return n / m;
}
```

Once we have a function in code, we can use it like in math using the application construct seen in a previous lecture:

Syntax:

<name> `(` <value> `[` `,` <value> `]*` `)`

(A name followed by an opening parenthesis followed by zero or more values separated by commas, followed by a closing parenthesis.)

Semantics:

when evaluating an expression of this form, the function designated by the name on the right is invoked using the values between parentheses as input arguments; when it completes execution, its return value becomes the value of the entire expression.

For example:

```
int y = f(3);  // stores 4 in y

double z = g(1, 2); // stores 0.5 in z
```

## Choice of types for functions

When encoding from math, we encounter the issue that Java data types do not map exactly to the common sets in calculus. Instead the following approximation are commonly used:

| Set in maths | Type in Java, C, C++ |
|---|---|
| $\mathbb{N}$‘ | `int` (or `byte`, `short`, `long`) |
| $\mathbb{Z}$‘ | `int` (or `byte`, `short`, `long`) |
| $\mathbb{Q}$‘ | `double` (or `float`) |
| $\mathbb{R}$‘ | `double` (or `float`) |
| $\mathbb{B}$‘ | `boolean` |

## Functions with effects

In languages like Java we have statements that “do” things in addition to simple computations, like printing values to screen (`System.out.println` etc.).

It is possible to use these statements inside a function, which means the function can compute a value *and* also perform some effects. Consider for example:

```
final float VAT_RATE 21.0; // value added tax

float computeVat(float price)
{
     float result = price * VAT_RATE / 100;
     System.out.printf("price = %.2f, vat = %.2f\n",
                       price, result);
     return result;
}

void start()
{
     float p1 = 320;
     float t1 = computeVat(p1);
     float p2 = 441;
     float t2 = computeVat(p2);
     float p3 = 112;
     float t3 = computeVat(p3);
     System.out.printf("Total VAT: %.2f\n", t1+t2+t3);
}
```

When run, this program will enter the function `computeVat` three times, and every time the effects inside it will be carried out. So the program prints:

```
price = 320.00, vat = 67.20
price = 441.00, vat = 92.61
price = 112.00, vat = 23.52
Total VAT: 183.33
```

With regards to terminology, we say that a function that contains effect-performing statements is an *effectful function*.

## Procedures: "functions" *with* effects, *without* value

Say you have a program where some effect is repeated two or more times. For example:

```
Scanner in = new Scanner(System.in);

String n1 = in.next();
System.out.printf("Hi %s!\n", n1);
String n2 = in.next();
System.out.printf("Hi %s!\n", n2);
String n3 = in.next();
System.out.printf("Hi %s!\n", n3);
```

As usual, as per the DRY rule we should abstract the common pattern in its own function. We can easily see the common parameter (the Scanner `in`), and the common effect (`next()` followed by `printf()`). However there is no mathematical value being computed, in other words the mathematical image of the function is not defined!

So what we really want to write looks like this:

```
/* nothing */ greet(Scanner in)
{
    String name = in.next();
    System.out.printf("Hi %s!\n", name);
    /* no return */
}

// (... later on ...)

Scanner in = new Scanner(System.in);
greet(in);
greet(in);
greet(in);
```

This is very well possible, however the syntax for a function definition is very strict: there must be *something* in the position before the name of the function.

For this purpose, languages in the C/C++/Java family have a special keyword: "`void`". *The "void" keyword is used every time there is a syntax position for a type in the language, but no type makes sense in that position.*

This is particularly the case for the function above:

```
void greet(Scanner in)
{
    String name = in.next();
    System.out.printf("Hi %s!\n", name);
    /* no return */
}
```

Such a function that performs some effects but does not compute a result is called a *procedure* in most programming language. The word "subroutine" is also used in some books but is quite old-fashioned.

## Vocabulary

In this lecture we have seen in turn:

- "simple" functions with no effects, that mirror mathematics; these are called *pure functions*;
- functions that contain some effectful statements, also called *effectful functions*, further divided in:
  - functions that have some effects and also compute a value (no special name), and
  - functions that have some effects and do not compute a value, also called *procedures*, in Java denoted with the keyword "`void`" instead of a return type.

The terms "pure function", "effectful function" and "procedure" are very general and used throughout all programming languages.

Meanwhile, as we have seen before there is another concept looming on the horizon, that of "objects" as in "object-oriented programming". If you remember the previous lecture, Java is part of a family of languages that provides construct for objects, in particular classes.

Classes, in object-oriented programming, can define their own "local functions" that make a function specific to a group of objects. So on top of the distinction above, a distinction is often made between:

– "global" functions that are applicable in general, independently from classes; and – "local" functions within a specific class. These are called *methods*.

For example in Python:

```python
def sayhi():
    # this function is global
    print("Hi!")

class Hello:

    def sayhi(self):
        # this function is local
        print("Hi!")
```

Most language have objet orientation as an optional feature, so a programmer can use global functions as long as no class is needed. Unfortunately, Java makes object orientation mandatory, which means that *all functions in Java are local to a class, ie. all functions must be methods*.

Because of this specific feature, people using only Java usually confuse the words "function" and "method": they would not know any better, because they never have seen a function that was not also a method! But do not be intimidated by this conflation, the terms are really distinct and you would do yourself a favour by remembering the difference.

Any way, the example above really should be placed in their class context, so we should really have written the following:

```java
class Hello
{

  void greet(Scanner in)
  {
    String name = in.next();
    System.out.printf("Hi %s!\n", name);
  }

  void start()
  {
    Scanner in = new Scanner(System.in);
```

```
    greet(in);
    greet(in);
    greet(in);
  }

  // ... a main() function somewhere here ...
}
```

In this code we have a class named "Hello" that defines two methods, one procedure called "greet" and another procedure called "start".

## Arguments are passed by value

There is a peculiarity of a lot of programming languages that tends to throw off some beginner programmers:

```
void foo(int x)
{
    int y = x - 3;
    System.out.printf("foo %d %d\n", x, y);
    x = x + 2;
}

void start()
{
    int x = 123;
    int y = x - 10;
    foo(x);
    System.out.printf("start %d %d\n", x, y);
}
```

What does this program print? Try it out before reading further.
The answer most beginners will propose is the following:

```
foo 123 120
start 125 120
```

However, the correct answer is the following:

```
foo 123 120
start 123 113
```

What is happening here? The key idea that you need to remember is that in Java, *the values of parameters that are received by a function body are copies of the values in the caller function*. Notice the word "copies"! So no original. This means that the original variable is left unchanged in the caller function (`start` in the example) during the computation of the called function (`foo` in the example). Moreover, the variables declared locally in each function (eg. `y` in `foo`) are just this, "local", so they are different variables from the variables with the same name in other functions (eg. `y` in `start`).

This idea is summarized by saying "arguments are passed by value" and "local variables are not shared between functions". This is a characteristing of Java and some other programming languages, and it does not hold in others. Be sure you pay attention to this feature when you learn a new language.

## Important concepts

- the mathematic concepts around a function, in particular *parameter*, *domain*, *expression*, *image*
- how to transcribe a mathematical function into Java;
- the common mappings of mathematical sets into Java data types;
- the role of "`return`";
- the difference between a *pure* and non-pure function;
- the difference between a non-pure function and a *procedure*;
- the role of "`void`";
- the meaning of the word *method*;
- how arguments to functions are passed by value and how to use this fact when reading/analysing a program.

## Further reading

- Introduction to Programming, chapter 4 up to and including 4.3.4 (pp. 135-151)
- Think Java, chapter 6 up to and including 6.3 (pp. 55-60), chapter 3 (pp. 25-38)

---

## Copyright and licensing

Copyright © 2014, Raphael 'kena' Poss. Permission is granted to distribute, reuse and modify this document according to the terms of the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.